Lab 2

Jordan Small
October 16, 2023

CDA3203 Computer Logic Design

Fall 2023

Dr. Maria Petrie
Florida Atlantic University

Section 1: The Half Adder

1.1: Handwork

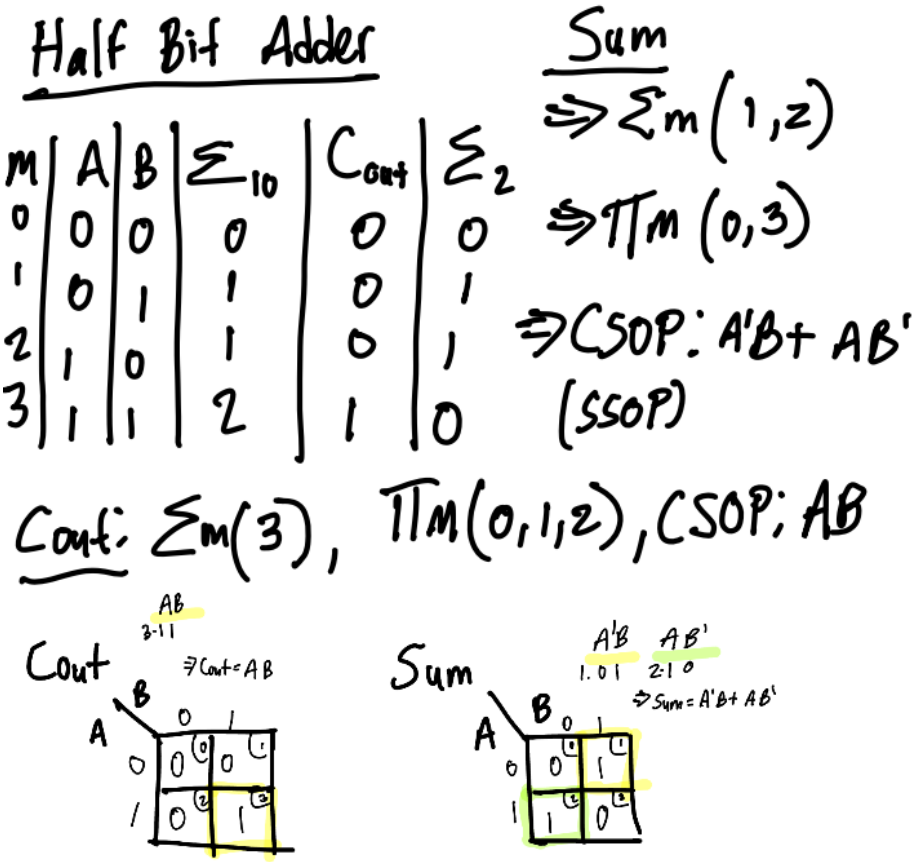


Figure 1a: The truth table for the half adder includes all possible 2-bit inputs and their respective sums, represented by two 1-bit outputs: the sum and the carry-out. The two combined outputs represent the sum in base 2; also included is the sum in base 10 to make the component's function very clear. Using the truth table, I formed K-maps and from them deduced the simplest sum of products for both outputs.

1.2: Diagram

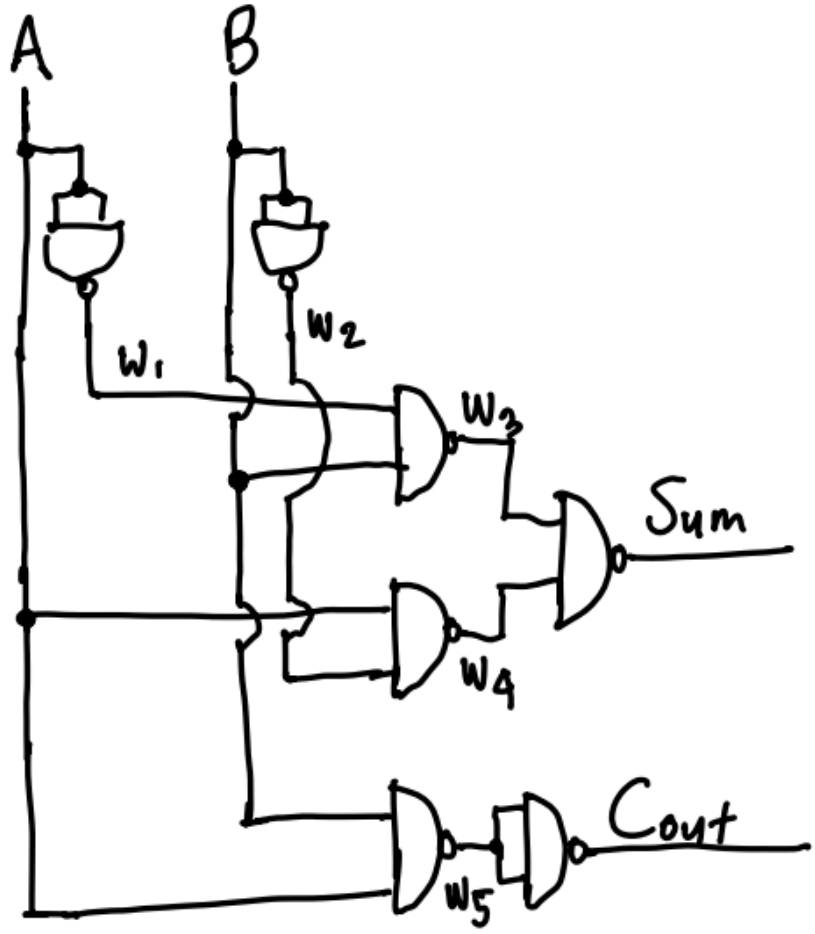


Figure 1b: Using the NOT-AND-OR equivalent diagram (formed from the simplest sum of products found above), I was able to construct an all-NAND equivalent diagram. Displayed are the inputs, outputs, and internal wires needed to make the circuit function.

1.3: Quartus

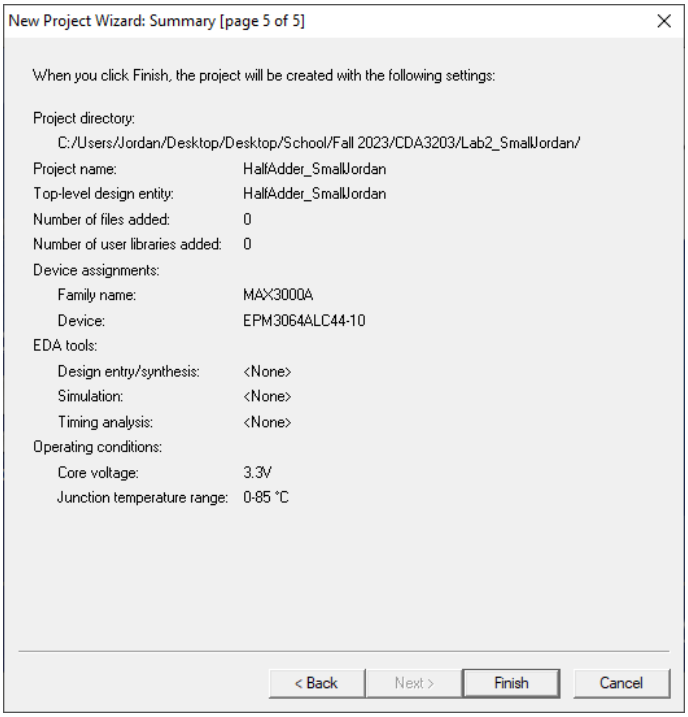
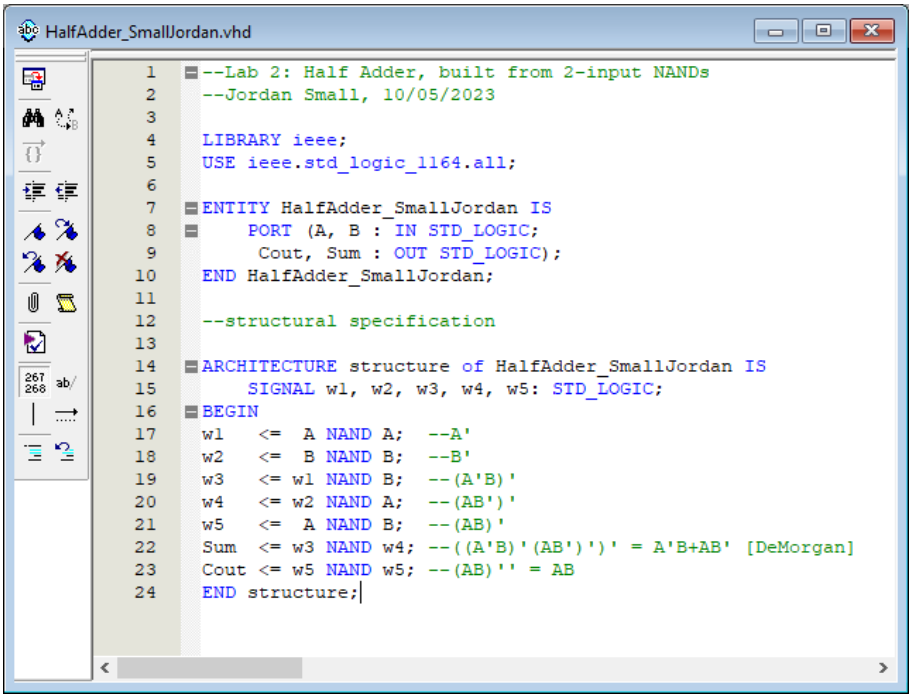


Figure 1c & 1d: Above is a snippet of the project settings, showing the root directory, device, and project name. Below is the VHDL code for the Half Adder circuit using the all-NAND equivalent: showcasing the internal wires needed to make the circuit function.



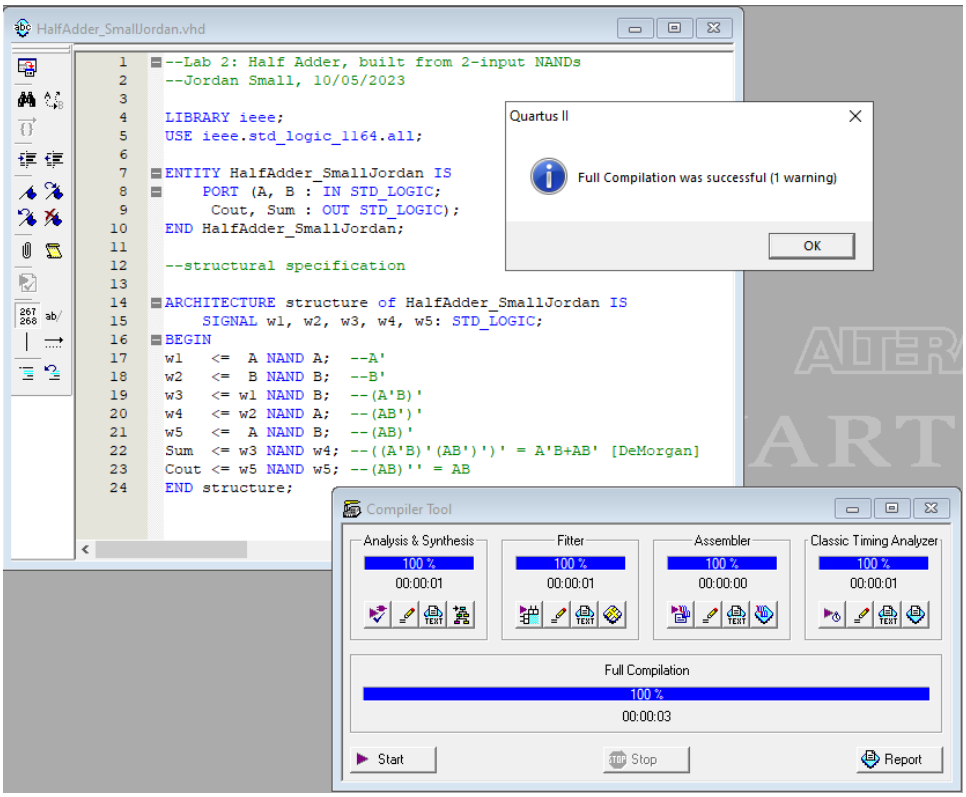


Figure 1e: The successful compilation (with no errors) of the circuit's VHDL code shows that there is no syntax error and that all connections are valid.

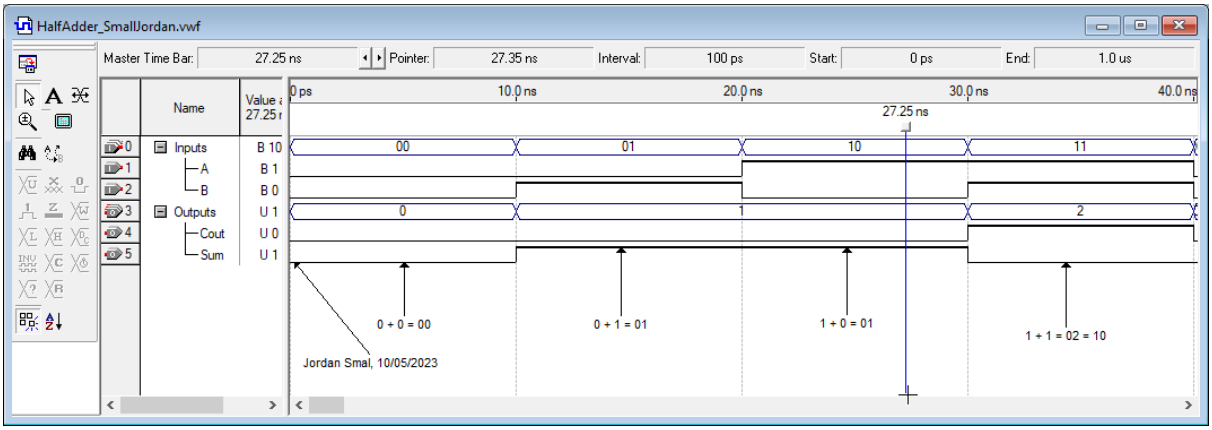


Figure 1f: Above is the timing diagram for the Half Adder—displaying all of the possible 1-bit binary additions and their correct, respective 2-bit results—highlighting the intended functionality of the circuit.

Section 2: The Full Adder

2.1: Handwork

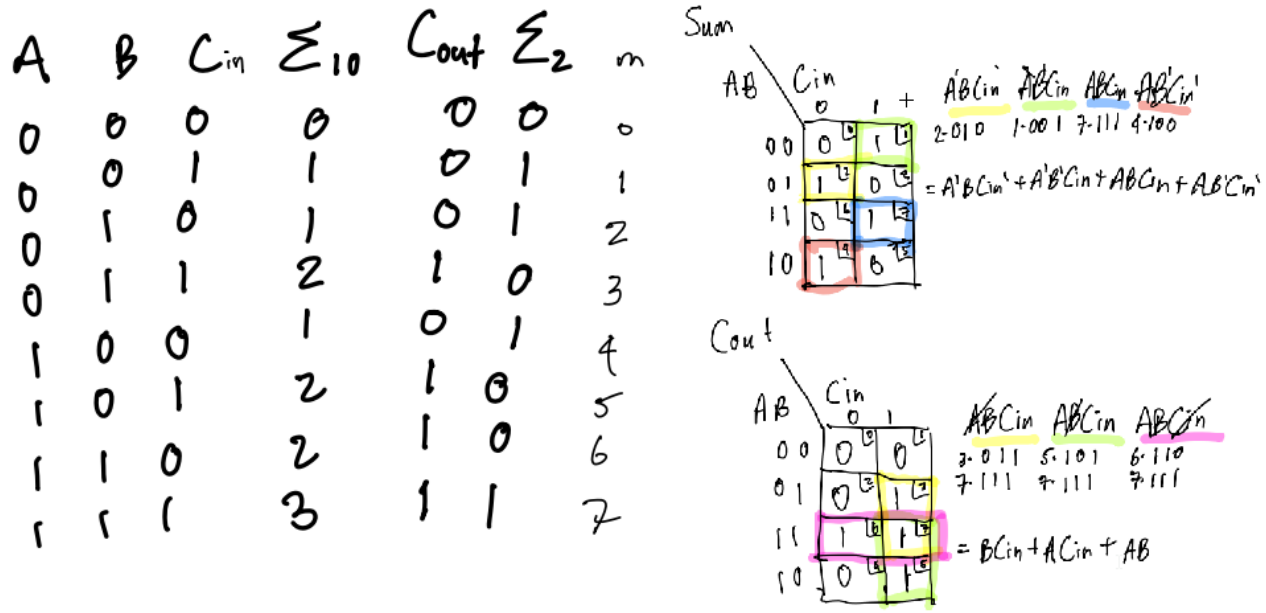


Figure 2a: The truth table for the full adder includes all possible 1-bit additions, including a carry-in value (the carry from the previous column) and their respective sums, represented by two 1-bit outputs: the sum and the carry. The two combined outputs represent the sum in base 2; also included is the sum in base 10 to make the component's function very clear. Using the truth table, I formed K-maps and from them deduced the simplest sum of products for both outputs.

2.2: Diagram

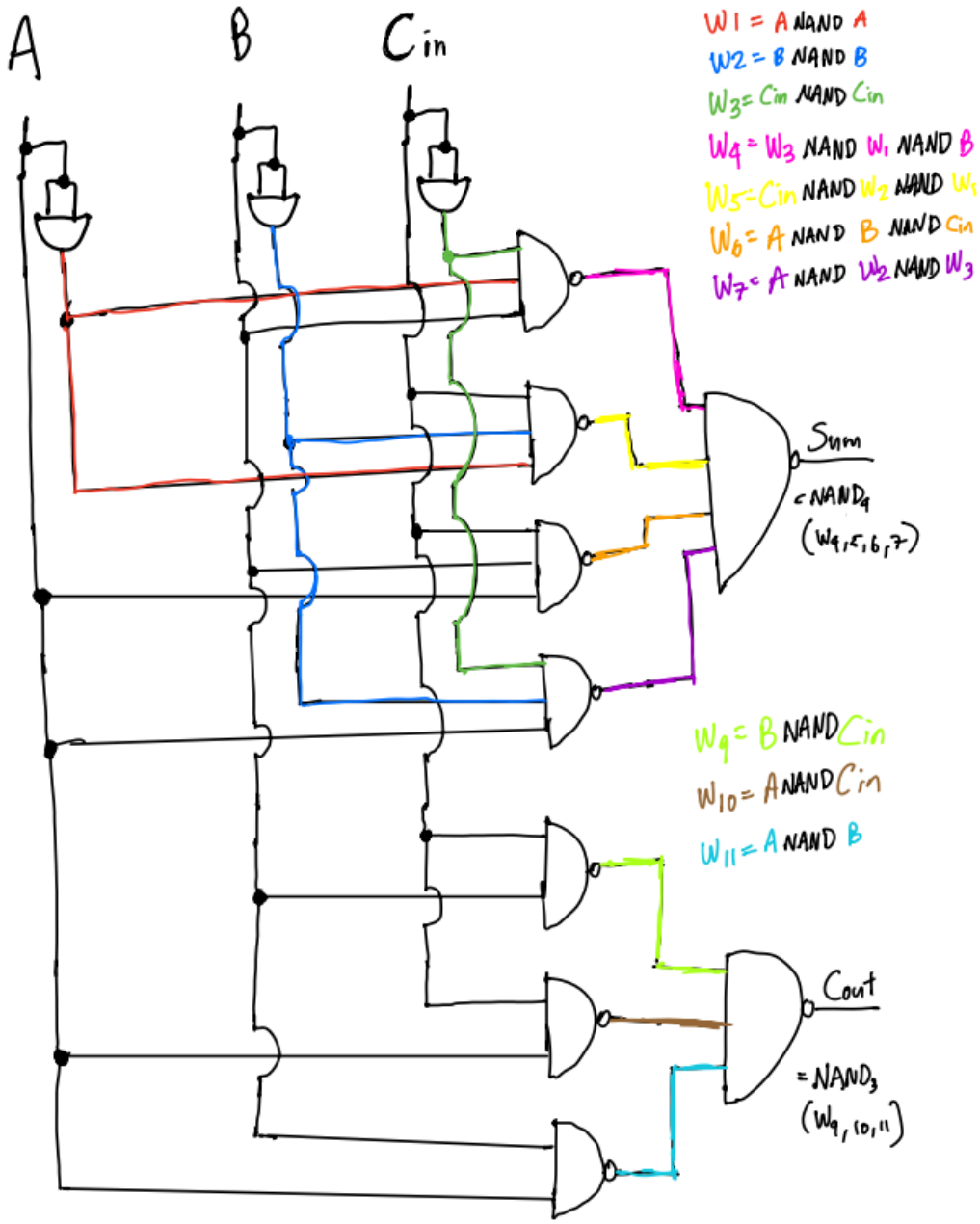


Figure 2c: Before drawing the diagram above, one was formed using a NOT-AND-OR from the simplest sum of products for both outputs. Since both sums included products from either three and/or four inputs, it was easiest to form the all-NAND above diagram using 3-input and 4-input NAND gates that were coded in a previous lab. Displayed as well are internal wires used to make the circuit function and their relations to input and output wires.

2.3: Quartus

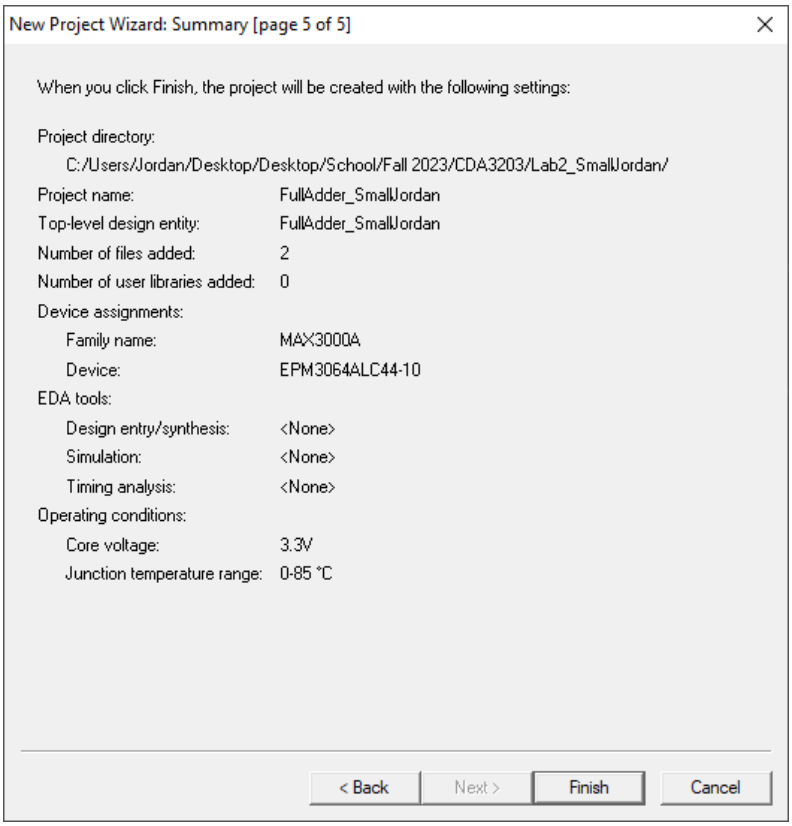


Figure 2d: Above are the project settings for the Full Adder including the root directory, project name, and device used. One thing to note in particular is the line that shows that there were two files added. For this project, the 3-input and 4-input NAND gates from Lab1 were utilized and included as components.

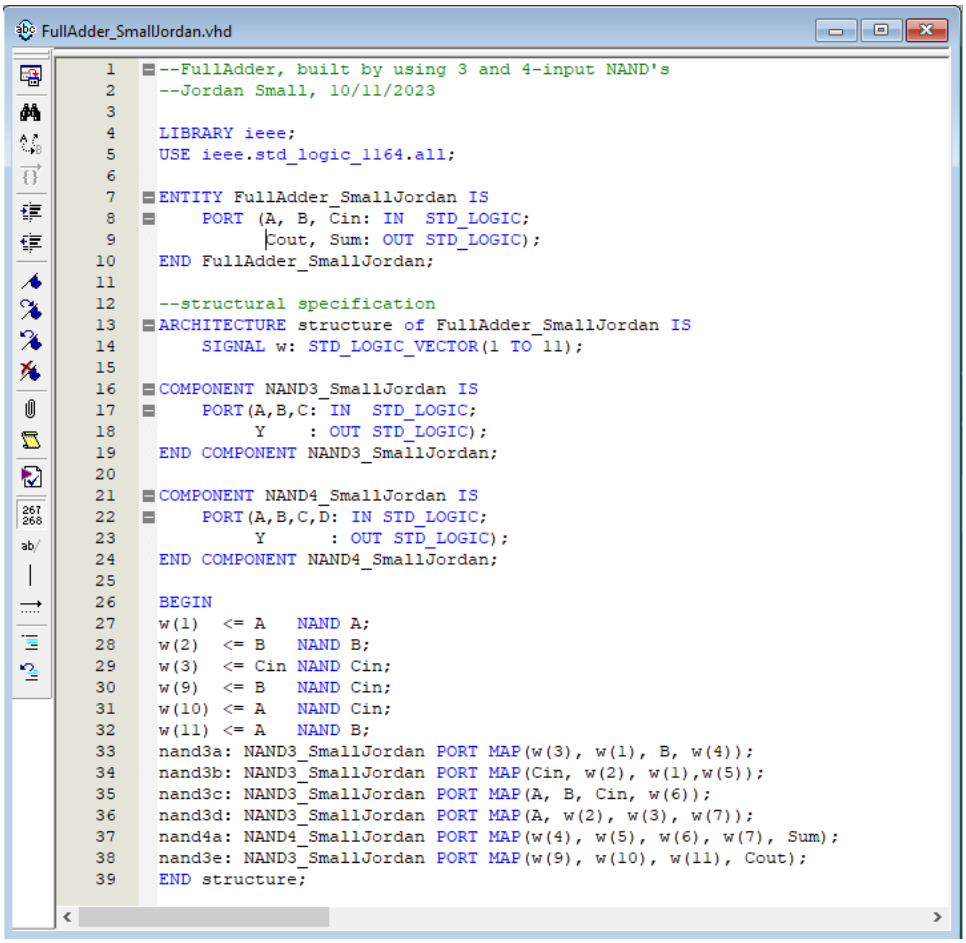
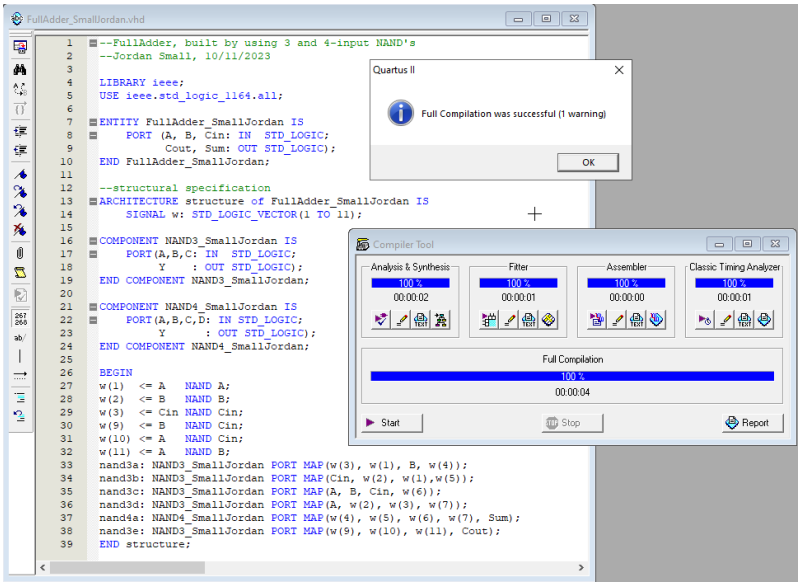


Figure 2e & 2f: Above is the VHDL code for the Full Adder circuit. The wire relations from figure 2d are declared and used in the aforementioned implementation of previous lab components: the 3-input and 4-input NAND gates. Line 14 shows the utilization of STD_LOGIC_VECTOR assignment, rather than individual pin assignment. Below is the code's successful compilation showing no errors.



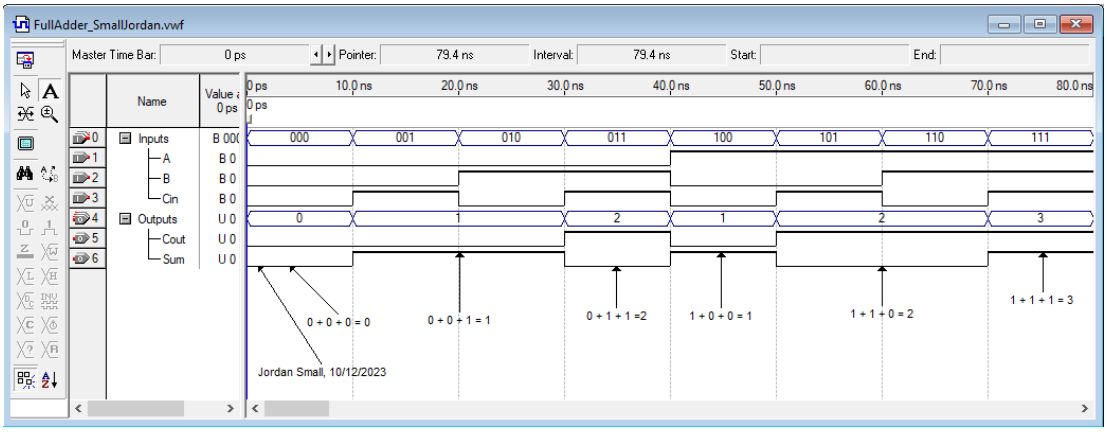


Figure 2g: The timing diagram for this circuit displays all possible 1-bit additions, including a carry-in value (the carry from the previous column) and their respective sums, represented by two 1-bit outputs: the sum and the carry. Annotations show that the additions were correct and the circuit functions as it should.

Section 3: The 4-bit Adder

3.1: Diagram and Handwork

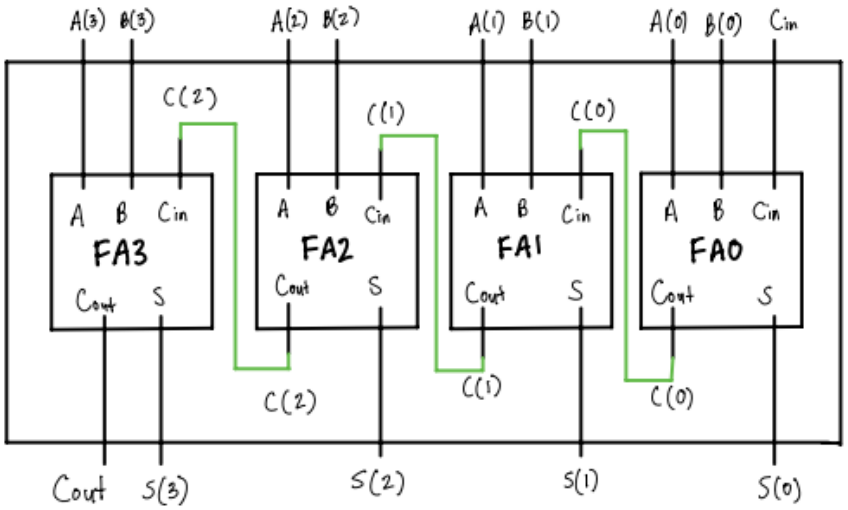


Figure 3a: The diagram above shows a 4-bit adder, built from using four 1-bit adders, displaying how the inputs and outputs relate to each other. Three internal wires were needed to use the carry-out value from the previous adder as the carry-in value for the next. The component adds two 4-bit binary numbers and a 1-bit carry-in and results in a 4-bit sum and a 1-bit carry out.

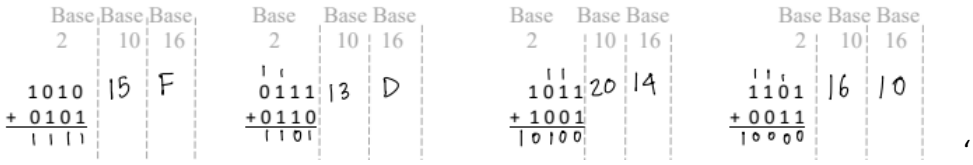


Figure 3b: Above are some examples of binary addition and their respective sums in base 2, 10, and 16. Later in this section, the two values being added will be assigned to A and B inputs as arbitrary values during a timing simulation to test the adder's functionality.

3.2: Quartus

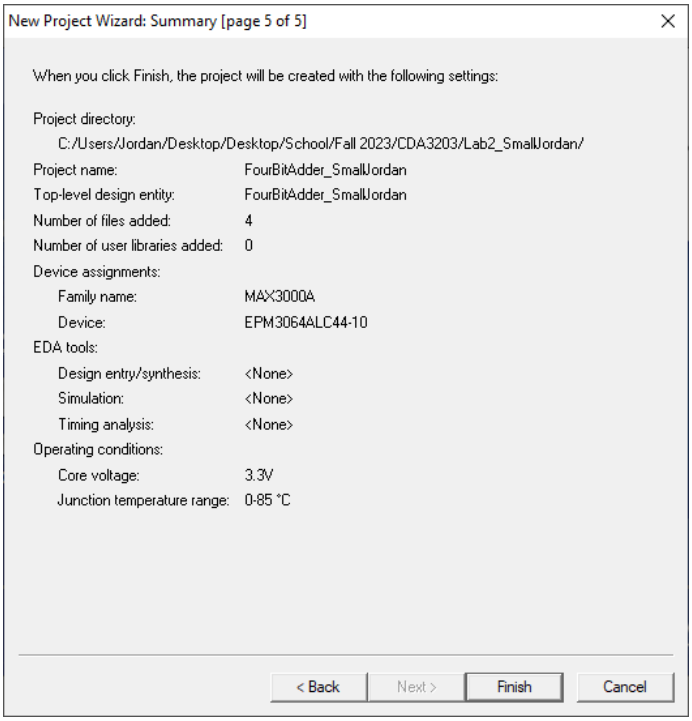


Figure 3c: Above are the project settings for the 4-bit adder, including the root directory, project name, and device used. One thing to note in particular is the line that shows that there were four files added. For this project, the 3-input and 4-input NAND gates from Lab1 were utilized and included as components. Also included are the previous circuit (the full adder), as well as a package I created, named "adders_SmallJordan," used to hold all of the adder components I would make for easier referencing in the larger adders that will be introduced later in this report.

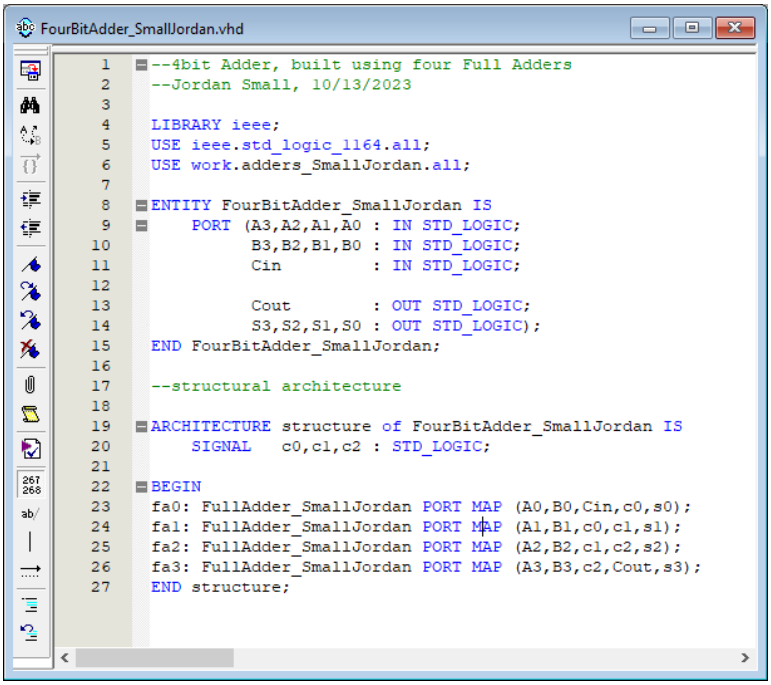


Figure 3d: Above is the VHDL code for the 4-bit adder. Depicted by figure 3a, the code uses four full adders and their relation to each other to form one 4-bit adder. These relations can be observed by the respective port mapping on lines 23-26. Likewise stated in figure 3a, the result will be a 4-bit binary number (represented by s3 down-to s0) and a 1-bit carry-out. On line 7, you can see the introduction of the "adders_SmallJordan" package—a package used to hold all the adder components for easier referencing in the larger adders that will be introduced later in this report.

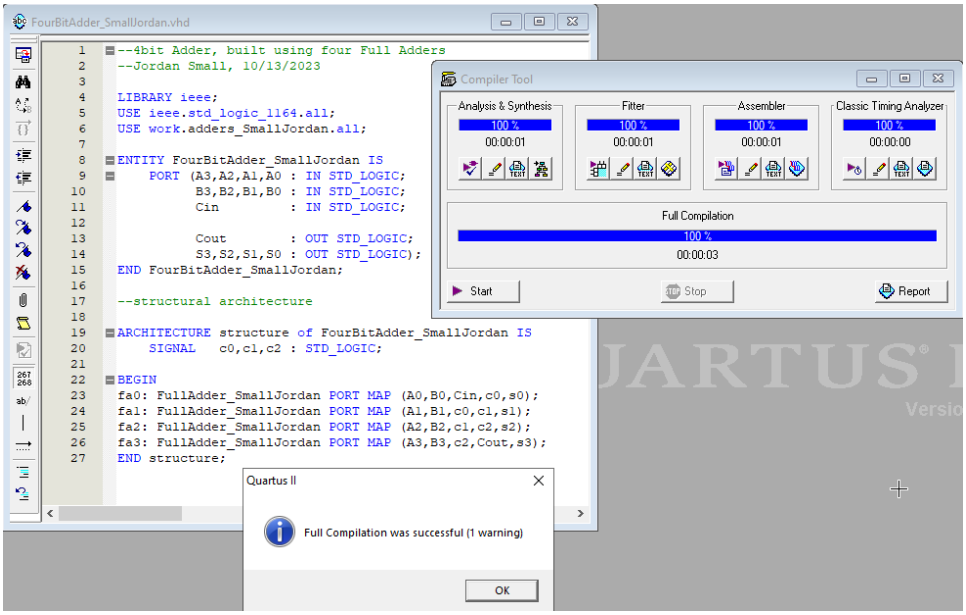


Figure 3e: Shown above is the successful compilation of the VHDL from figure 3c—showing no errors.

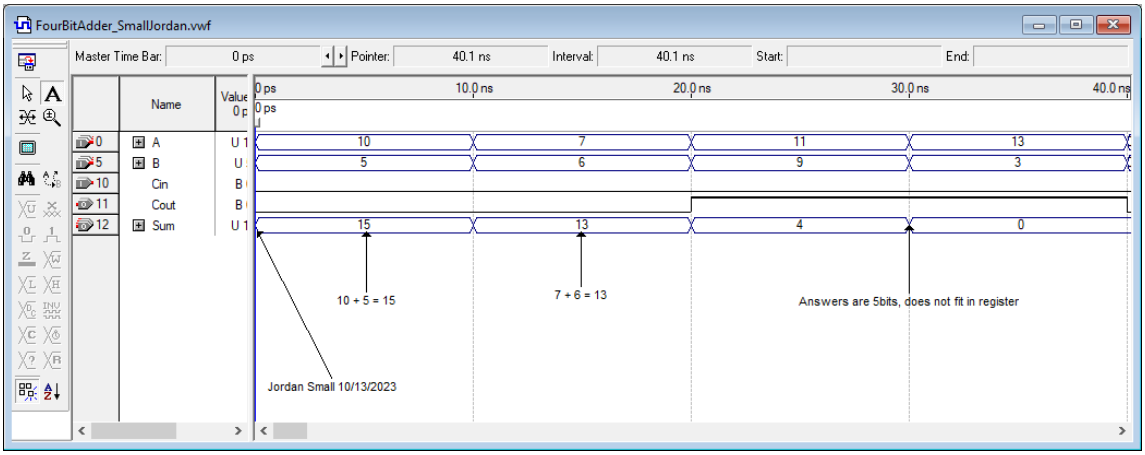


Figure 3f: Using the values displayed in figure 3b as arbitrary values for A and B inputs (as well as the value of Cin to be forced low), the timing diagram acts as a test to see that the values were being added correctly. The annotation highlights two things: 1) for the first two tests, the adder performed correctly and displayed the expected results (shown in decimal value, to make the addition clear). 2) for the second two tests, the results were 5bits and therefore did not fit in the register; a carry-out value of 1 reinforced this outcome.

Section 4: The 8-bit Adder

4.1: Diagram

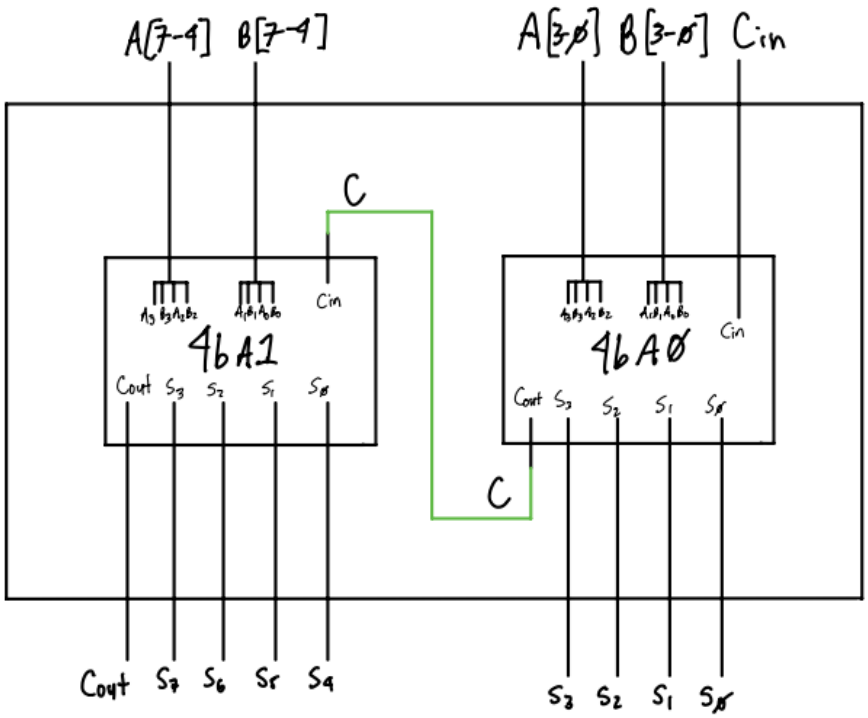


Figure 4a: The diagram above shows an 8-bit adder, built from using two 4-bit adders, displaying how the inputs and outputs relate to each other. One internal wire, C, was needed to use the carry-out value from the previous adder as the carry-in value for the next. The component adds two 8-bit binary numbers (assume the bracket notation for A and B inputs represent an array, i.e A[3-0] represents A3, A2, A1, A0; this notation will be used in later diagrams throughout this report) and a 1-bit carry-in and results in an 8-bit sum (represented by s7 down-to s0) and a 1-bit carry out.

4.2: Quartus

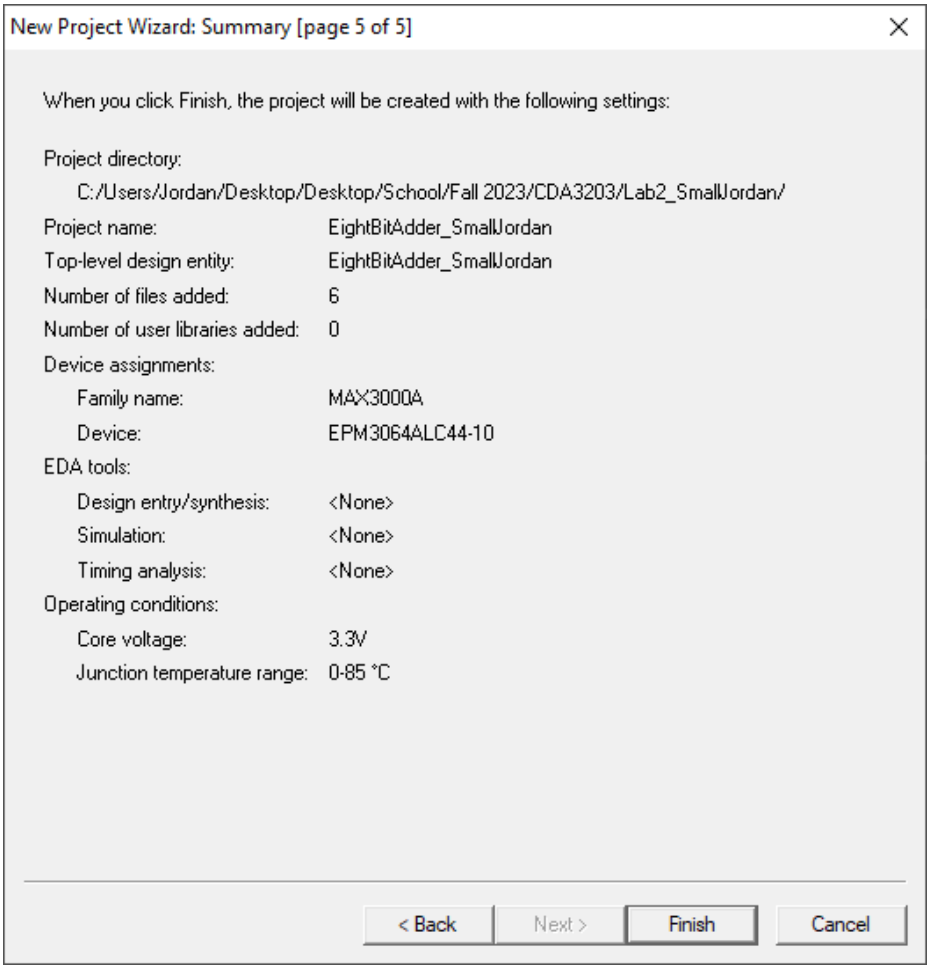


Figure 4b: Above are the project settings for the 8-bit adder, including the root directory, project name, and device used. One thing to note in particular is the line that shows that there were six files added. For this project, the 3-input and 4-input NAND gates from Lab1 were utilized and included as components. Also included are the three previous adder components from this lab, as well as the previously mentioned "adders_SmallJordan" package.

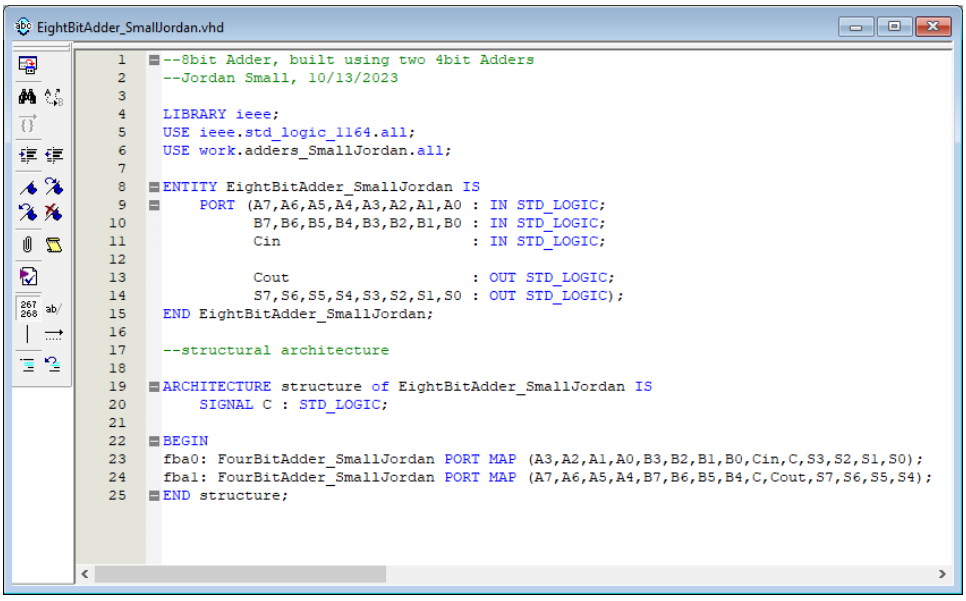


Figure 4c: Above is the VHDL code for the 8-bit adder. Depicted by figure 4a, the code uses two 4-bit adders and their relation to each other to form one 8-bit adder. These relations can be observed by the respective port mapping on lines 23 and 24. Likewise stated in figure 4a, the result will be an 8-bit binary number (represented by s7 down-to s0) and a 1-bit carry-out. Line 6 shows the “adders_SmallJordan” package implementation.

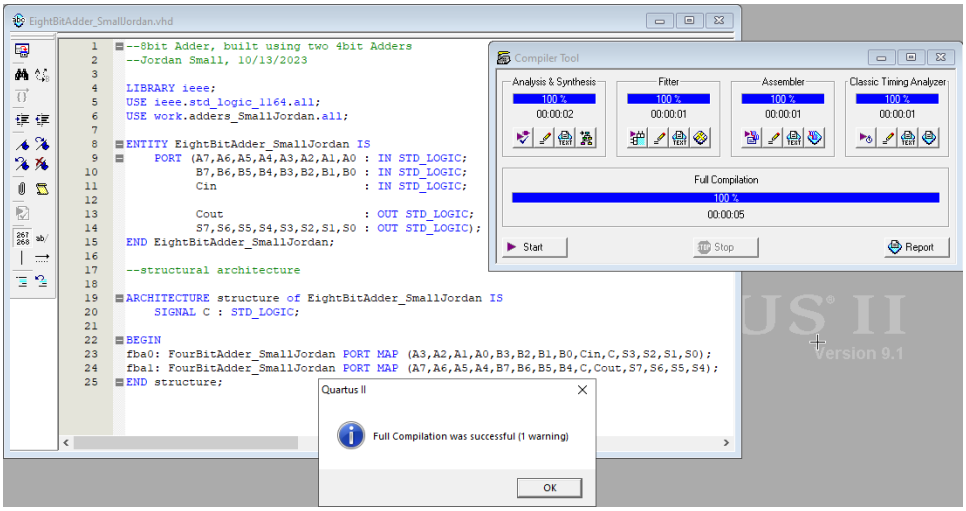


Figure 4d: Above is the successful compilation—without errors— of figure 4c.

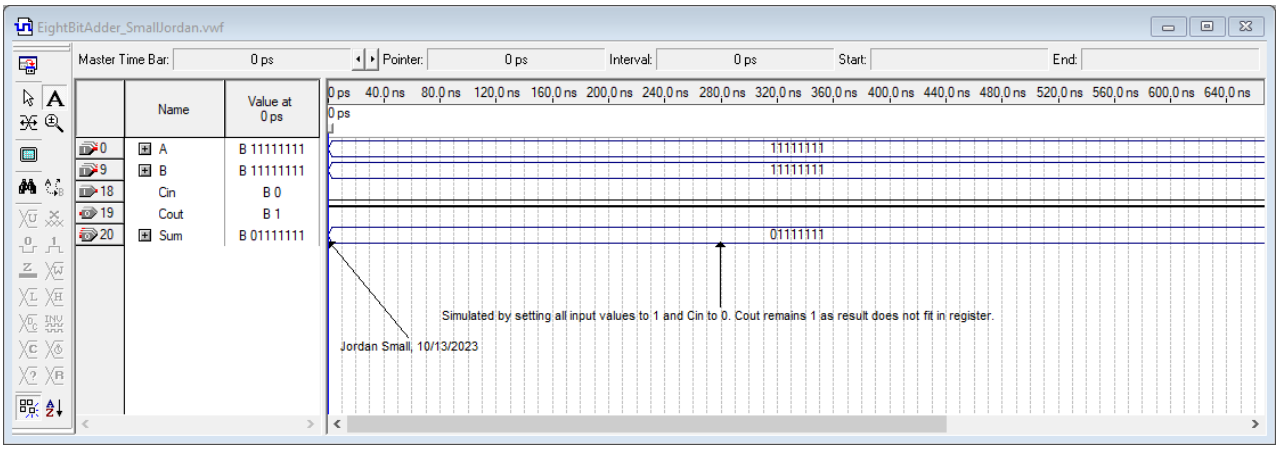


Figure 4e: To test this circuit's functionality, a simulation was done forcing all input values to high and Cin to low. The result, as expected, does not fit in the register and shows a constant Cout value of 1 throughout the simulation, reinforcing the aforementioned.

Section 5: The 16-bit Adder

5.1: Diagram

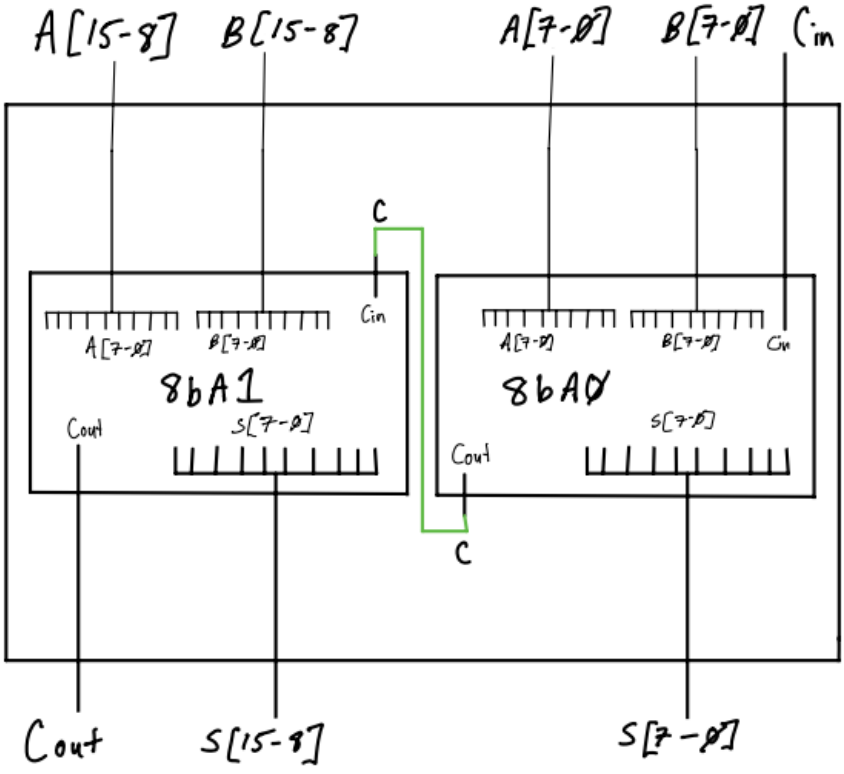


Figure 5a: The diagram above shows a 16-bit adder, built from using two 8-bit adders, displaying how the inputs and outputs relate to each other. One internal wire, C, was needed to use the carry-out value from the previous adder as the carry-in value for the next. The component adds two 8-bit binary numbers (using the same pin notation from figure 4a) and a 1-bit carry-in and results in a 16-bit sum (represented by the same notation from figure 4a) and a 1-bit carry out.

5.2: Quartus

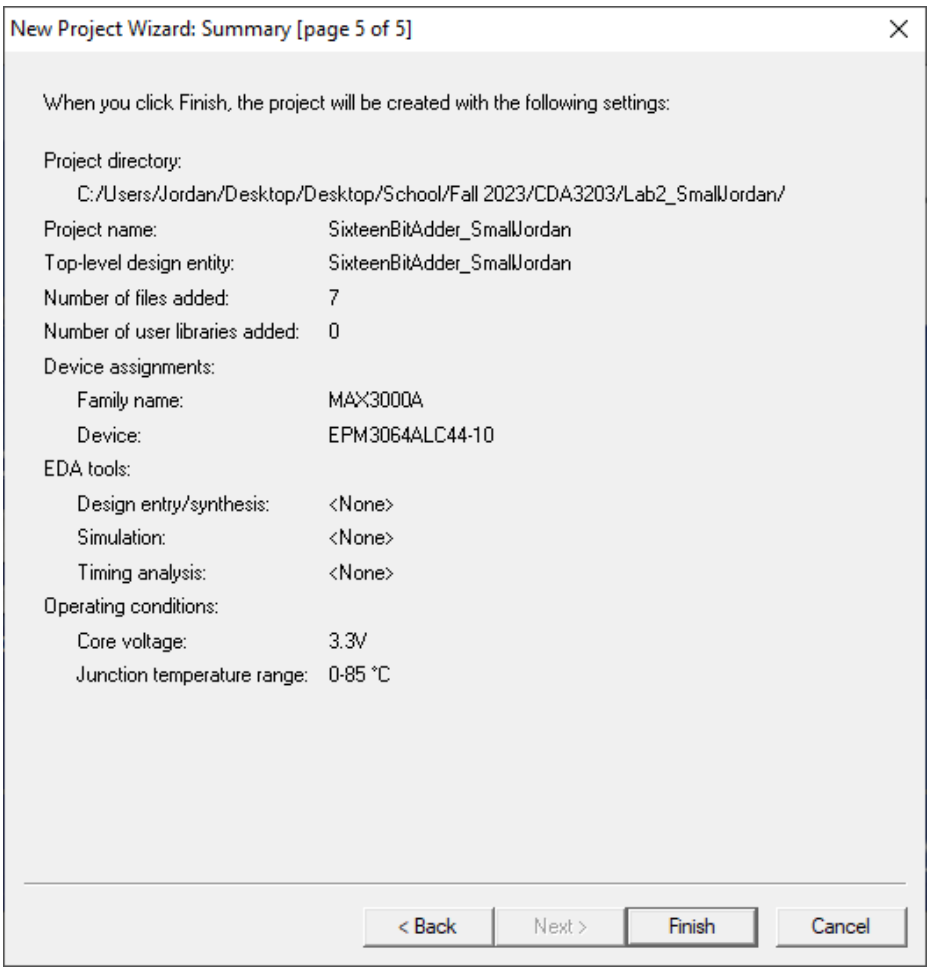


Figure 5b: Above are the project settings for the 16-bit adder, including the root directory, project name, and device used. One thing to note in particular is the line that shows that there were seven files added. For this project, the 3-input and 4-input NAND gates from Lab1 were utilized and included as components. Also included are the four previous adder components from this lab, as well as the previously mentioned “adders_SmallJordan” package.

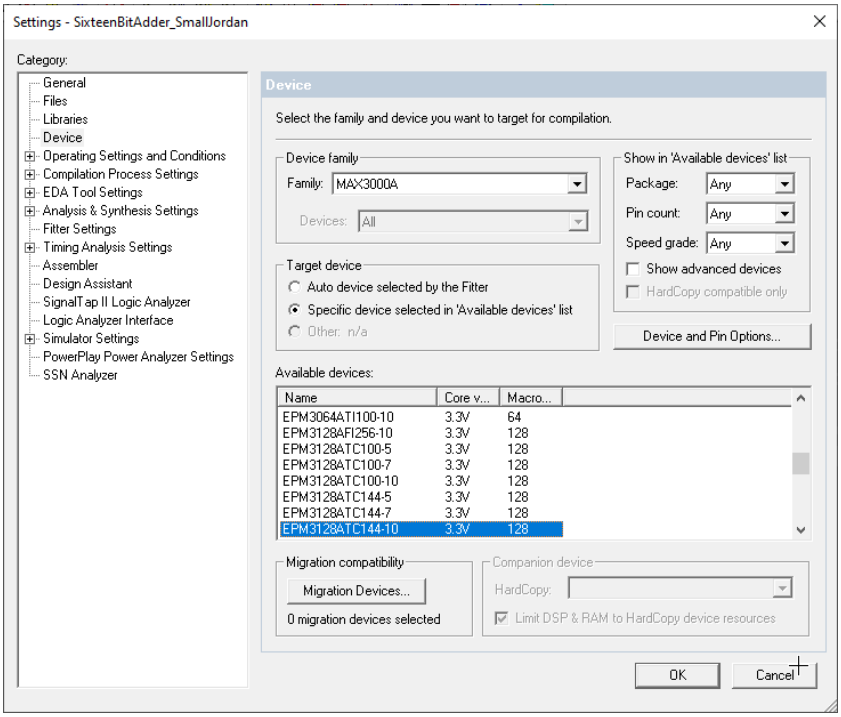


Figure 5c: Shortly after starting the project, I was running into some issues compiling the VHDL (shown later in this report). Early-on in the troubleshooting process, I realized that there were not enough pins available in the device used in figure 5b and thus I selected a new device (highlighted in the figure) with more pins.

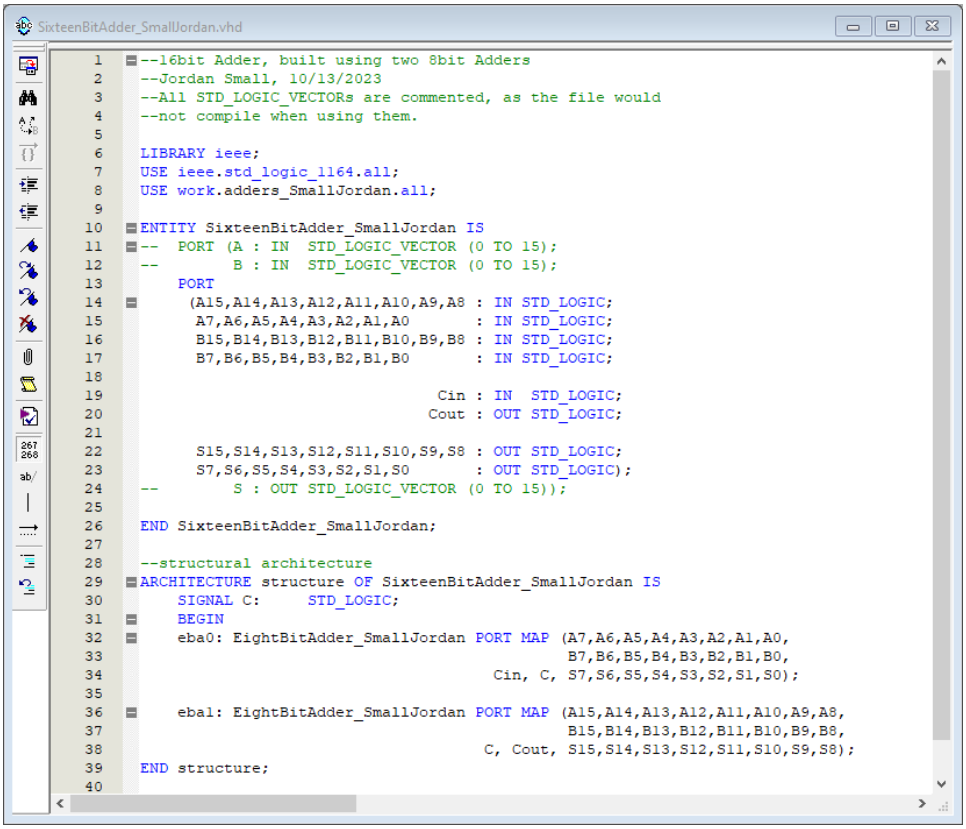


Figure 5c: Above is the VHDL code for the 16-bit adder. Depicted by figure 5a, the code uses two 8-bit adders and their relation to each other to form one 16-bit adder. These relations can be observed by the respective port mapping on lines 32 through 38. Likewise stated in figure 5a, the result will be a 16-bit binary number and a 1-bit carry-out. Line 8 shows the "adders_SmallJordan" package implementation. As mentioned by comments on lines 3 and 4, lines 11, 12, and 24 were commented-out, as the code was having a hard time compiling when using vectors to declare pins. The errors that occurred stayed consistent with variable declarations or the lack thereof, namely due to vectors (the code compiled fine without them).

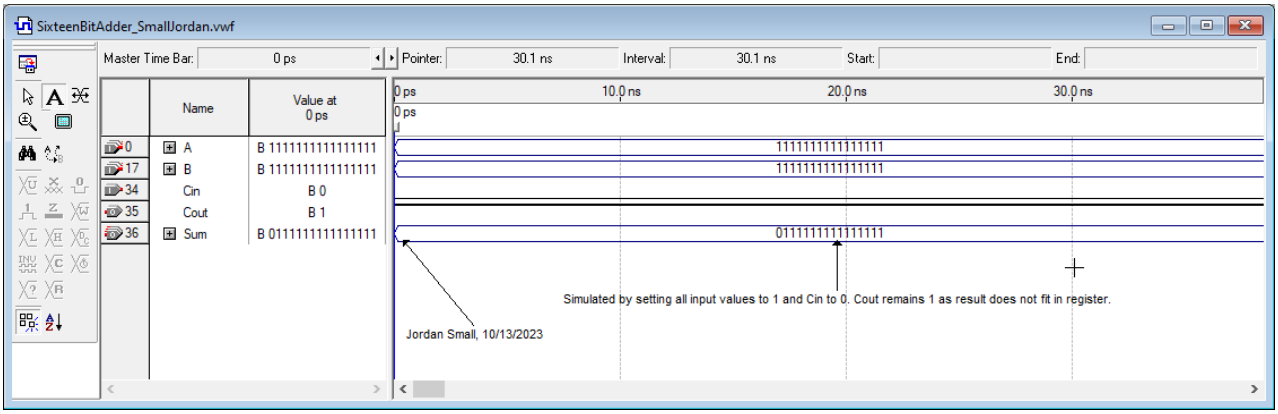


Figure 5d: To test this circuit's functionality, a simulation was done forcing all input values to high and Cin to low. The result, as expected, does not fit in the register and shows a constant Cout value of 1 throughout the simulation, reinforcing the aforementioned.